12th European Conference on Object-Oriented Programming
Workshop on Parallel Object-Oriented Scientific Computing
Brussels, Belgium, July 20-24, 1998

Applying Fortran90 and Object-Oriented Techniques to Scientific Applications

Charles D. Norton[1], Viktor K. Decyk[1,2], and Joan Slottow[3]

1 Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA
2 Department of Physics and Astronomy, UCLA, Los Angeles, CA, USA
3 Office of Academic Computing, UCLA, Los Angeles, CA, USA

I. Introduction

High-performance parallel computing is having a profound impact on the size and complexity of physical problems which can be modeled. This impact has been a long time in coming, because the learning curve in adapting to this new world of computing is steeper than was imagined. Nevertheless, more and more success stories have convinced computational scientists that parallel computing is important and is here to stay. To model complex, 3D physical systems, older paradigms of programming which were adequate in a 2D world, now become cumbersome and limiting.

One needs a language with higher levels of abstraction for such problems. A few pioneering computational scientists have turned to the object-oriented paradigm, and particularly C++ for help. Fortran90, while not a true object-oriented language, also has powerful facilities for abstraction, and object-oriented programming is possible by emulating in software the OO features which are not in the language [1-2]. These features allow the code to be designed using the same abstractions that exist in object-oriented languages, but in a Fortran framework more familiar to most computational scientists. In this paper, we will discuss our experience in using Fortran90 for parallel adaptive mesh refinement, scientific visualization, and plasma particle-in-cell simulations.

II. Parallel AMR

One area we have been working on is parallel, unstructured adaptive mesh refinement (AMR). Organizing and programming the data structures for parallel AMR is very difficult. The main structure is the computational mesh that represents a complex geometry with many components. With the older style programming paradigms available in Fortran77 and C, using parallel AMR is so complex that its use has been limited. Object-oriented methods using C++, have been applied to manage the complexity for this problem before [3]. We have found that Fortran90 is equally useful.

For example, a Fortran90 module allows user-defined data types and related routines to be encapsulated in a class - an important feature of object-oriented programming. A mesh_module (abbreviated) can contain the definition of a mesh data structure and mesh operations:

```
module mesh_module
use mpi_module ; use heapsort_module
implicit none
private
public :: mesh_create_incore, mesh_repartition, &
             mesh_visualize
type mesh
   private
   type(node), dimension(:), pointer :: nodes
   type(edge), dimension(:), pointer :: edges
   type(element), dimension(:), pointer :: elements
   type(b_element), dimension(:), pointer ::boundary_elements
end type mesh
contains
subroutine mesh_create_incore(this, mesh_file)
   type(mesh), intent(inout) :: this
   character(len=*), intent(in) :: mesh_file
   ! details omitted
end subroutine mesh_create_incore
   ! additional member routines
end module mesh_module
```

This encourages the development of simple interfaces whose internal features can be changed without impacting their usage in the main program. Features of one module can be made available to another via the "use" statement. Fortran90 also supports pointer structures, in addition to many other dynamically allocatable structures. The example above illustrates pointers to dynamic arrays. With these features AMR objects can be created easily.

Sometimes it is necessary to interface to programs written in other languages. The ParMeTiS mesh partitioner [4] is used for parallel mesh partitioning, yet it is written in the C programming language. Interlanguage communication between Fortran90 and C is not a problem, once the proper format for function references is determined. Our design uses a single routine that acts as the conduit between the Fortran90 parallel AMR code and the C mesh partitioner:

```
subroutine mesh_repartition(this)
type(mesh), intent(inout) :: this
   ! statements omitted
   call PARMETIS(mesh_adj, mesh_repart, nelem,nproc) ! C call
   call mesh_build(this, new_mesh_repart=mesh_repart)
end subroutine mesh_repartition
```

The graph description of the distributed mesh is passed by reference to the C code which returns the partitioning to Fortran90. Since Fortran90 optional arguments can be selected by keyword, rebuilding the mesh using the new partitioning can reuse the same code as constructing the original mesh. Since the mesh_build routine is private to the module containing the mesh_repartition routine, mesh_build cannot be called by the main program.

Such features allow for a very abstract design and representation of source code for parallel AMR. A main program that loads a mesh, distributes it among the parallel processors, creates the mesh data structure, performs repartitioning and visualization, now looks like this:

```
program pamr
 use mesh_module
 implicit none
 ! statements omitted
 type(mesh) :: in_mesh
    call MPI_INIT(ierror)
    call mesh_create_incore(in_mesh, in_file)
    call mesh_repartition(in_mesh)
    call mesh_visualize(in_mesh, "visfile.plt")
    call MPI_FINALIZE(ierror)
end program pamr
```

Our parallel AMR library routines have been applied to the finite-element simulation of electromagnetic wave scattering in a waveguide filter, and long-wavelength infrared radiation in a quantum well infrared photodetector as test cases in two-dimensions (a three-dimensional system is under development). The software currently runs on the Cray T3E, HP/Convex Exemplar, IBM SP2, and Beowulf-class "pile-of-pc's" running the LINUX operating system [5].

III. Scientific Visualization

In the days when the world was 2D, many computational physicists included graphics in the output of their simulation by making use of graphics libraries. In the new 3D world, the amount of expertise required to embed visualization into simulations is so great, that the computational scientists have largely abandoned this altogether, and do visualization in an independent, post-processing step. This leads to a lot of time-consuming busy work in moving data around and learning the concepts and particulars of some visualization program. It is not unusual in our group for a student to spend an entire summer learning how to make one beautiful picture of a complex simulation. It also makes interactive exploration of simulations nearly impossible. Finally, the underlying visualization programs themselves can change or disappear in response to the market place, causing further problems for the scientist.

Our goal was to remove these barriers to visualization and to provide a stable environment for the scientific programmer. Our approach was to develop a Fortran90 class library which produced a typical 3D graphical image for each particular type of data [6]. We call our class library Visual Data Objects (VDO) because its objects describe the structure of the underlying scientific data and are self-visualizing. The objects are designed to always produce a reasonable image by default, and optional controls are provided so that the scientist can customize or improve the images if desired. The kinds of data objects currently supported are those most common: 1D, 2D, and 3D vector and scalar fields on regular meshes, although support for irregular objects is planned. The Fortran90 class library contains objects which specify the data structure and aspects of the visualization, information the user needs to control. Only a few lines of code need to be added to the program to obtain an image.

Those commercial and public-domain visualization programs which can be called from an application program are used as the underlying visualization engines. These programs generally have a C interface and are sometimes difficult to call from Fortran. To more easily interface with these visualization engines, a C++ library was written to drive them. This library allows us to change the underlying engine without impacting the Fortran scientific code. For example, VDO currently supports IBM's Data Explorer as a visualization engine. We plan to support the Visualization ToolKit (VTK) in the future because it is free and can run on all platforms. Thus, if IBM's product becomes unavailable to the scientist, he or she can request that one of the other visualization engines be used.

There is one C++ visualization class in VDO for each visualization engine, but the Fortran program need not be concerned with how it operates. The Fortran classes delegate to the C++ classes the procedures directly relating to the graphics, while performing the remaining ones itself. In fact, a single routine (Visualize) acts as the conduit between the Fortran90 and C++ code. Objects in Fortran90 and C++ are generally not compatible. For example, Fortran90 pointers are hidden structures and C++ objects can contain hidden pointers. However, by restricting what goes into an object, we can safely pass a Fortran90 derived type that is read in C++ as a struct and is used in a special constructor to recreate the Fortran90 object in C++ [6].

Visual Data Objects was designed to be run from parallel programs. Since many installations require that long-running parallel programs be run in batch mode, VDO is designed to either display the graphical images or write them to files at the user's discretion. More information is available on the web page:
http://computing.oac.ucla.edu/sciviz/visualdataobjects.htm

IV. Plasma Particle-in-Cell Simulations

For the past several years, we have been using both Fortran90 and C++ for plasma simulation. Our first effort was a plasma particle-in-cell (PIC) simulation code written in C++, using message-passing on distributed memory parallel computers. Once the new language features of Fortran90 became known to us, we translated this original C++ code into Fortran90 and discovered, somewhat to our surprise, that most of the C++ code could be translated directly, and the performance of the Fortran90 code was substantially better, typically by a factor of two [7-8]. This encouraged us to examine Fortran90 further.

Development of the parallel PIC in both languages has continued, with ideas suggested by one language being incorporated into the other. In general, we have found strengths and weaknesses in both languages, discussed in the next section. Our future plans are to develop a flexible, modular set of classes and objects which could be used to quickly assemble a new code targeted for a particular problem domain and architecture.

V. Conclusions

For most purposes, Fortran90 has proved very useful in complex, scientific programming. The use of derived types and dynamic memory and pointers permitted complex structures to be expressed easily and safely. In general, we have found that Fortran90 is a safer language for programming than C++ and development is faster, since the compiler finds more errors. This is achieved by restricting or encapsulating use of some language features, such as pointers, but results in reduced flexibility.

Perhaps the most notable area where reduced flexibility might be important is the lack of language support for inheritance, which allows related types to share data and procedures. This lack was addressed by developing emulation techniques for inheritance and run-time polymorphism [1-2]. (These features will be supported in Fortran2000.) It turned out that for most of our scientific calculations, inheritance was not used very much neither in Fortran90 nor in C++. Even when it was used, it was typically for the non-scientific parts of the problem, such as in performance monitoring or timing.

Run-time polymorphism was occasionally useful, but not always with inheritance. For example, in the PIC simulations, 1D, 2D, and 3D models have similar organizations and interfaces, but the algorithms and data for each model are not simply related. We would like to be able to decide at run time, what kind of model we are running. This desired behavior was implemented in Fortran90 by creating polymorphic types that could refer to anything we liked. This was also possible to do in C++, by creating an abstract interface class which contained no data members or methods, and then have other classes inherit nothing but the interface. Most object-oriented languages combine inheritance and run-time polymorphism into a single language mechanism. The fact that inheriting nothing is useful indicates that these concepts probably should be separated.

References

[1] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "Expressing Object-Oriented Concepts in Fortran90 " ACM Fortran Forum 16, 13 (1997).

[2] V. K. Decyk, C. D. Norton, and B. K. Szymanski, NASA Tech Briefs, vol. 22, no. 3, p. 100 (1998). See also: http://www.cs.rpi.edu/~szymansk/oof90.html

[3] Shephard, M., Flaherty, J., de Cougny, H., Ozturan, C., Bottaso, C., and Beall, M. (1995). Parallel automated adaptive procedures for unstructured meshes. In Parallel Computing in CFD, number R_807, pages 6.1-6.49. Agard, Neuilly-Sur-Seine, France.

[4] Karypis, G., Schloegel, K., and Kumar, V. (1997). ParMeTiS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 1.0. Technical report, Dept. of Computer Science, U. Minnesota.

[5] J. Z. Lou, C. D. Norton, and T. Cwik, "A Robust Parallel Adaptive Mesh Refinement Software Package for Unstructured Meshes," submitted for publication, 1998.

[6] J. Slottow and V. K. Decyk, "Visual Data Objects -- A Visualization Tool for the Scientist Programmer," submitted for publication, 1998.

[7] C. D. Norton, "Object Oriented Programming Paradigms in Scientific Computing," Ph. D. Thesis, Rensselaer Polytechnic Institute, Troy, NY, 1996.

[8] C. D. Norton, B. K. Szymanski, and V. K. Decyk, "Object-Oriented Parallel Computation for Plasma Simulation," Comm. of ACM, vol 38, no. 10, p. 88 (1995).